UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Tomaž Štih

Jedro YX za ZX Spectrum

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJ (VSŠ)
RAČUNALNIŠTVA IN INFORMATIKE, SMER PROGRAMSKA OPREMA

MENTOR: doc. dr. Tomaž Dobravec

Ljubljana, 2016

UNIVERSITY OF LJUBLJANA

FACULTY FOR COMPUTER AND INFORMATION SCIENCES

Tomaz Stih

The YX Kernel for ZX Spectrum

BACHELOR'S THESIS

PROFESSIONAL STUDY PROGRAMME -
SOFTWARE ENGINEERING

THESIS ADVISOR: doc. dr. Tomaž Dobravec

Ljubljana, 2016

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:
Tematika naloge: Jedro YX za ZX Spectrum

V nalogi razvijte preprosto mikrojedro operacijskega sistema za okolje z zelo omejenimi viri. Jedro naj vsebuje vsaj naslednje funkcionalnosti: osnovno upravljanje s pomnilnikom, upravljanje s procesi, mehanizme medprocesne komunikacije (sinhronizacija procesov in sistemski klici) ter en primer gonilnika naprave.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Tomaž Štih, vpisna številka 24013372, avtor pisnega zaključnega dela študija z naslovom: Jedro YX za ZX Spectrum

IZJAVLJAM

1. da sem pisno zaključno delo študija izdelal samostojno pod mentorstvom doc. dr. Tomaža Dobravca;

2. da je tiskana oblika pisnega zaključnega dela študija istovetna elektronski obliki pisnega zaključnega dela študija;

3. da sem pridobil vsa potrebna dovoljenja za uporabo podatkov in avtorskih del v pisnem zaključnem delu študija in jih v pisnem zaključnem delu študija jasno označil;

4. da sem pri pripravi pisnega zaključnega dela študija ravnal v skladu z etičnimi načeli in, kjer je to potrebno, za raziskavo pridobil soglasje etične komisije;

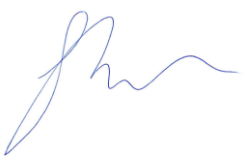
5. soglašam, da se elektronska oblika pisnega zaključnega dela študija uporabi za preverjanje podobnosti vsebine z drugimi deli s programsko opremo za preverjanje podobnosti vsebine, ki je povezana s študijskim informacijskim sistemom članice;

6. da na UL neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve avtorskega dela v elektronski obliki, pravico reproduciranja ter pravico dajanja pisnega zaključnega dela študija na voljo javnosti na svetovnem spletu preko Repozitorija UL;

7. dovoljujem objavo svojih osebnih podatkov, ki so navedeni v pisnem zaključnem delu študija in tej izjavi, skupaj z objavo pisnega zaključnega dela študija.

V Londonu, dne 15. avgusta 2016

Podpis avtorja



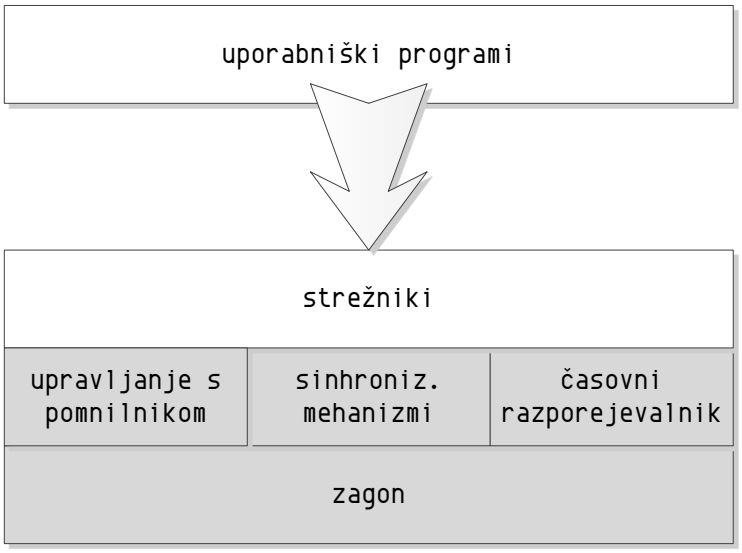
Kazalo

Seznam uporabljenih kratic

Kratica	angleško	slovensko
RAM	Random Access Memory	Bralno-pisalni pomnilnik
ROM	Read Only Memory	Bralni pomnilnik
UART	Universal Asynchronous Receiver/Transmitter	Specializirani čip za serijsko komunikacijo
KB	Kilobyte	Kilobajt oz. 1024 bajtov
HDMI	High Definition Multimedia Interface	Vmesnik za prenos avdiovizualnih nestisnjenih digitalnih podatkov
IM	Interrupt Mode	Prekinitveni način
RTS	Ready to Send	Signal za pripravljenost za pošiljanje podatkov
CTS	Clear to Send	Signal za pripravljenost za sprejemanje podatkov
DTE	Data Terminal Equipment	Podatkovna terminalna naprava
IPC	Inter-Process Communication	Medprocesna komunikacija (tudi: sinhronizacijski mehanizmi)

Povzetek

V diplomskem delu smo razvili sodobno mikrojedro za izvirno različico Mavrice[1] z 48 kilobajti pomnilnika iz leta 1982. Njegove ključne lastnosti so: upravljanje s pomnilnikom, predkupna večopravilnost, sinhronizacijski mehanizmi (dogodki, upravljanje s prekinitvami) in upravljanje s strežniki, prek katerih uporabniški programi dostopajo do funkcij jedra (protokoliziranje sistemskih klicev) ter drugih registriranih strežnikov.



Slika 1. Nivojska struktura jedra.

Ključne odločitve pri načrtovanju sta narekovali velikost pomnilnika, hitrost procesorja (točneje ... majhnost pomnilnika in počasnost procesorja) ter tudi želja, da bi bilo mogoče jedro prenesti na druge platforme. Logično kompleksnejši deli mikrojedra so zato napisani v programskem jeziku C, kritični pa v zbirniku Z80.

> *Dave Cutler kot eno največjih napak pri razvoju VMS omenja to, da ga niso razvili v visokonivojskem jeziku. [1]*

[1] Imena računalnikov se ponavadi ne prevajajo. Mavrica je izjema, saj je bil to v osemdesetih letih 20. stoletja široko sprejet naziv za računalnik ZX Spectrum, ki se je redno uporabljal v knjigah in drugih publikacijah, od Mojega Mikra do Cicibana. [2]

Za razvoj je uporabljen prevajalniški paket SDCC za ANSI C 11.

Mikrojedro je izjemno optimirano, nekateri deli so bili napisani večkrat. Prevedeno zasede le 6 kilobajtov

Ključne besede: Diplomska naloga, mikrojedro, operacijski sistemi, predkupna večopravilnost, mikroprocesorji, vgradni sistemi.

Poglavje 1 Uvod

Cilj diplomske naloge je bil razvoj sodobnega mikrojedra v okolju z zelo omejenimi viri. Za to okolje smo izbrali Mavrico, izumrli mikroračunalnik iz osemdesetih let 20. stoletja.

> *Mavrico poganja 8-bitni procesor Z80 s hitrostjo 4 Mhz. Naslovi lahko 64 kilobajtov pomnilnika, od tega je prvih 16 kilobajtov bralnega, 48 kilobajtov pa bralno-pisalnega. TV-modulator zna generirati sliko ločljivosti 256 x 192 pik, za vsakih 8 x 8 pik pa je možno določiti barvo črnila in ozadja. 1-bitni brenčač omogoča sintezo poljubnih zvočnih učinkov in je omejen le s hitrostjo procesorja. Za strojne razširitve je na voljo priključek Edge, ki omogoča dostop neposredno do glavnega vodila.*

Programerju takšno okolje omogoča učenje zelo kompleksne snovi na zelo preprostih primerih. Osemdeseta so bila čas, ko je bil programer zelo blizu stroju, sodoben osebni računalnik marsikaj skrije pred njim.

Kdor bi se želel primerljivega podviga lotiti s sodobnim osebnim računalnikom, bi za spodoben rezultat potreboval nekaj let, naučil pa bi se manj.

> *Temeljni principi delovanja računalnikov so že desetletja enaki. Nekaj primerov: 1) Nekoč je zaporedna vrata krmilil program v realnem času, danes nas o istih dogodkih obvešča UART s prekinitvami, ki pred nami skrije dejansko dogajanje, 2) v sodobni tipkovnici prebiva mikrokontroler, ki pred nami skrije kompleksnost branja signalov z membrane in dekodiranja pritisnjenih tipk, 3) USB so v resnici nekoliko močneje (nivojsko) protokolirana zaporedna vrata, 4) VGA po zaslonu riše podobno, kot je Atari Pong risal na TV sprejemnik; namesto ene ima tri nožice -R, G in B-, na katerih ob pravem trenutku spremeni napetost in s tem povzroči izris pike na želenem mestu na zaslonu, in 5) HDMI ni nič drugega kot fantastično hiter zaporeden prenos surove slike.*

Poglavje 2 Razvoj, orodja in proces

Za Mavrico so na voljo številna razvojna orodja; tako sodobna kot tudi takšna, ki so na voljo le za izumrla okolja, denimo mikroračunalnike Z80 z operacijskim sistemom CP/M. Razen redkih izjem so skoraj vsa moderna okolja odprtokodna in prosta ter na voljo za Windows, Linux in Mac.

Za »profesionalno« *(kolikor je razvoj za 30 let staro platformo lahko profesionalen)* delo je uporabnih več prevajalnikov; emulatorjev, na katerih je mogoče preizkusiti delovanje kode, in razhroščevalnikov na nivoju zbirne kode. Razhroščevanje na nivoju kode C (ang. source level debugging) žal ni mogoče. Pomagamo si lahko tako, da prevajalniku za C naročimo, da poleg objektne kode proizvede še listing, iz katerega je razvidno, v kaj je prevedel posamične C–vrstice, in simbolno datoteko, ki vsebuje podatke o tem, na katerem naslovu se nahaja kateri C-simbol (fukcija, spremenljivka). Nato program razhroščujemo z razhroščevalnikom za zbirnik.

> *Celoten razvoj diplomske naloge je bil opravljen na računalniku z operacijskim sistemom Linux. Uporabljena so bila orodja emulator FUSE z vgrajenim razhroščevalnikom, prevajalnik za C SDCC in za delo z zbirnikom Z80 in standardni urejevalnik gedit, ki smo ga prilagodili razvoju za Z80 in okolju SDCC.*

Ker je bila velikost in hitrost posameznih delov jedra ključna, možnosti razhroščevanja pa omejene – smo se odločili, da pri programiranju kombiniram zbirnik in kodo C. Zbirnik smo uporabil tam, kjer sta bili hitrost in velikost kode ključni. In C tam, kjer je bila koda bolj zapletena in bi jo bilo v zbirniku težje brati in posledično nadzorovati ter razhroščevati.

2.1 SDCC

SDCC je kratica za Small Device C Compiler. Gre za še en odprtokodni prevajalnik, ki temelji na znamenitem Small-C, katerega kodo je leta 1980 *(kot je bilo tedaj v navadi)* kar v časopisu Dr. Dobb's objavil James E. Hendrix. Small-C je bil tudi osnova za prevajalnik GNU C.

Zasnova SDCC omogoča relativno preprosto prilagoditev povprečnemu mikrokontrolerju (ang. retargetable) in enostavnemu mikroprocesorju. Prevajalnik je v odprtokodni skupnosti dobro in široko podprt, je stabilen, podpira ANSI-standarde za jezik C: C89, ISO C99 in ISO C11 [7] in proizvaja anekdotično [9] mnogo bolj optimalno Z80 kodo od prevajalnika Z88DK.

2.2 Zavržena razvojna okolja

Pri pisanju diplome so bila nekatera manj primerna razvojna okolja tudi zavržena.. Med njimi velja omeniti prevajalnik za zbirnik za Mavrico Pasmo, ki ga žal ni mogoče uporabiti z nobenim C-prevajalnikom. Isti vzrok je onemogočil tudi uporabo prevajalnika za zbirnik GNU Assembler in razhroščevalnika GNU debugger, ki sta sicer najboljši obstoječi okolji.

> *Na podlagi GNU orodij je kot stranski produkt diplomske naloge nastala serija člankov: »So you want to be a ZX Spectrum developer,« ki so objavljeni na mojem blogu [10] in podrobneje opišejo vzpostavitev razvojnega okolja okolja za razvoj.*

In, nenazadnje, je bilo zavrnjeno razvojno okolje Z88DK, ker prevajalnik-C po kvaliteti optimizacije kode in podpornih C-standardov (še) ne dosega okolja SDCC.

2.3 Potek razvoja

Razvoj je potekal v nasprotju z ustaljenimi praksami; jedro je bilo pisano »z ljubeznijo«, brez rokov, načrtov in drugih »umetnih« omejitev. Iz orodjarne programskega inženirstva smo si izposodili le prakso *refactoringa*. Da smo kodo jedra lahko stisnili v pet kilobajtov, smo morali praktično vsak del napisati najmanj dvakrat. Vsaka naslednja iteracija je proizvedla optimalnejšo kodo in preprostejšo, čistejšo arhitekturo. Sedanje jedro je manjše od 5 kilobajtov. Ob pisanju besedila diplomske naloge razvijam knjižnico za povezavo z internetom in sodoben grafični uporabniški vmesnik, ki bosta jedro spremenila v pravi operacijski sistem; vse skupaj pa bo na koncu stisnjeno v 16 kilobajtov.

2.4 FUSE

Fuse je emulator Mavrice, ki zna oponašati vse znane modele Mavrice in veliko večino strojne opreme, ki je tedaj obstajala. Za razvoj je bilo ključno, da FUSE omogoča emulacijo osnovnega (in pri nas najbolj priljubljenega) modela ZX Spectrum 48K, vmesnika Interface 1 oz. serijske komunikacije prek RS-232 in najbolj razširjenega modela miši Kempston.

2.5 Repozitorij GIT

Za upravljanje z verzijami jedra smo uporabili orodje GIT, ki je eden izmed dveh priljubljenih odprtokodnih sistemov za upravljanje z verzijami (drugi je Subversion).

Koda je javnosti na voljo na portalu *Github*:

http://github.com/tstih/yx

Poglavje 3 Zagon sistema

Ob signalu za resetiranje procesor Z80 omogoči prekinitve ter začne izvajati program na naslovu 0x0000. Na Mavrici je to začetek bralnega pomnilnika (ang. ROM).

> *Testiranje je pokazalo, da Z80 registra **AF** in **SP** nastavi na vrednosti **0xffff**, vsebina drugih pa ni predvidljiva. Vendar tega obnašanja uradna dokumentacija ne opisuje.*

Prvi ukaz na večini sistemov z Z80 je ***DI*** (disable interrupts). Ponavadi namreč program prvih nekaj trenutkov izvajanja potrebuje »mir«, da lahko izvrši inicializacijo, in zato onemogoči prekinitve.

3.1 Prekinitve

YX deluje v prekinitvenem načinu IM1. V tem načinu se vsako petdesetinko sekunde pri obnavljanju zaslona zgodi prekinitev. Ob prekinitvi procesor shrani trenutno vrednost programskega števca na sklad in skoči na naslov ***0x38***.

Z80 pozna še prekinitev NMI , ki je v izvirni različici Mavrice ni mogoče sprožiti, vendar pa to funkcionalnost uporabljajo nekatere strojne razširitve. Ob nastopu prekinitve NMI program shrani trenutno vrednost programskega števca na sklad in skoči na naslov ***0x66***.

3.2 Ukazi RST

Ukaz ***RST nn*** (dovoljene vrednosti za nn so ***0x08***, ***0x10***, ***0x18***, ***0x20***, ***0x28***, ***0x30*** in ***0x38***) shrani vrednost trenutnega programskega števca na sklad in izvede skok na podprogram na absolutnem naslovu nn. Npr. ob ukazu ***RST 0x08*** skoči na absolutni naslov ***0x08***.

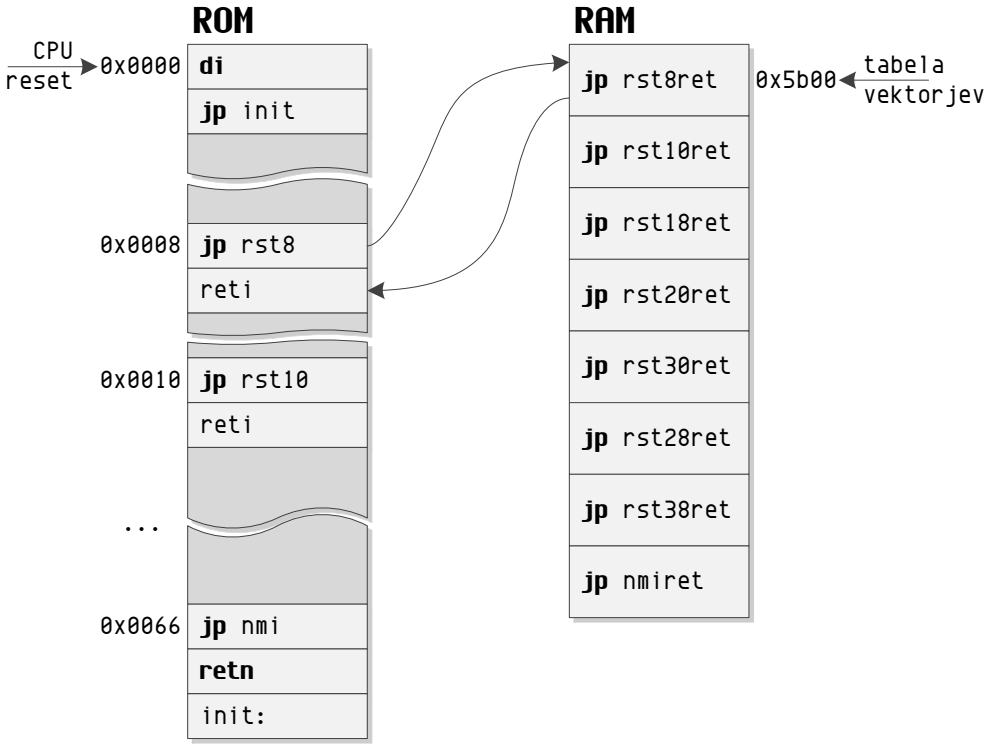
> *Čemu **RST** sploh obstaja? Uporabniški programi ga uporabljajo kot znani naslov za klicanje funkcij operacijskega sistema. Ker je instrukcija **RST** dolga le en bajt, je priročna tudi za prekinitvene točke pri razhroščevanju.*

3.3 Vektorji

Opisani prekinitvi (prekinitev 50 Hz IM1 in prekinitev NMI) ter ukazi ***RST*** za skok uporabljajo absolutne naslove v pomnilniku. To pomeni, da mora biti na omenjenih naslovih v bralnem pomnilniku ustrezen podprogram, saj ne bi bilo dobro, da bi ukazi skočili v neznano.

V žargonu YX tem posebnim naslovom na začetku pomnilnika pravimo vektorji. Ob zagonu sistema YX poskrbi, da so ustrezno nastavljeni. Hkrati jedro omogoča tudi prevezavo vektorjev, ki bi bili sicer zapečeni v bralnem pomnilniku in jih ne bi bilo mogoče več spreminjati. To je izvedeno z uporabo zrcalne tabele v bralno-pisalnem pomnilniku, ki se inicializira ob zagonu.

Naslednja slika prikazuje izvedbo vektorske tabele.



Slika 2. Izvedba vektorske tabele in njena prevezava v bralno-pisalni pomnilnik.

Na sliki je s puščicami označen primer, ko procesor izvede ukaz ***RST 0x08***. Na absolutnem naslovu ***0x0008*** je ukaz, ki skoči na tabelo vektorjev v bralno-pisalnem pomnilniku. Tabela se nahaja na naslovu ***0x5b00***.

V njej je za vsak vektor privzeto zapisan skok nazaj na naslednjo instrukcijo, tj. naslov ***0x0009***, kjer je ukaz ***RETI***. Vektor je dolg 3 bajte (opkoda jp + naslov skoka).

Vse privzete vrednosti v tabeli vektorjev vsebujejo skok na naslednji ukaz, od koder so bile klicane.

Za lažje prevezovanje YX vsebuje funkcijo intr_set_vect.

```
#define RST08   0
#define RST10   1
#define RST18   2
#define RST20   3
#define RST28   4
#define RST30   5
#define RST38   6
#define NMI     7

extern void intr_set_vect(void (*handler)(), byte vec_num);
```

3.4 Funkcija main()

Po vzpostavitvi tabele vektorjev zagonski program vzpostavi sklad za jedro, t. i. sistemski sklad. Prekopira inicializirane C-spremenljivke iz bralnega v bralno-pisalni pomnilnik (oz. iz DATA v segment BSS), znova omogoči prekinitve in skoči na funkcijo ***_main()***[2]. Primer preproste funkcije ***main()*** uporabniške konzole:

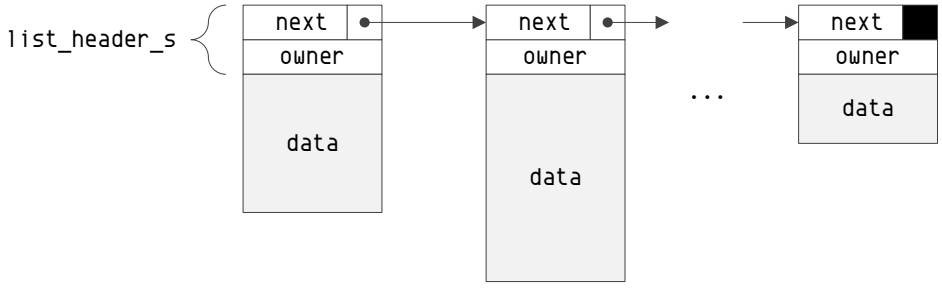
```
void main() {
        /* initialize operating system and user heap */
        mem_init(&sys_heap, SYS_HEAP_END - &sys_heap + 1);
        mem_init((void*)USR_HEAP_START, USR_HEAP_END - USR_HEAP_START + 1);

        /* scheduler RST38 */
        intr_set_vect(tsk_switch, RST38);

        /* install keyboard scanner */
        tmr_install(kbd_scan, EVERYTIME, SYS);

        /* shell, stack size=512 */
        tsk_create(shell, 512);
}
```

[2] Prevajalnik za C ob prevajanju v zbirnik pred imena funkcij avtomatično doda podčrtaj.

Poglavje 4 Upravljanje z viri

Pomnilnik, opravilo, časovnik ali dogodek so viri jedra YX. Upravljanje z viri je v YX poenoteno. Podatkovne strukture vseh virov so umeščene v sezname in vsebujejo osnovne meta podatke (npr. o lastniku bloka), YX pa vodi seznam vseh seznamov.

Upravljanje z viri je ena izmed temeljnih nalog vsakega jedra. [4] Tovrstno »računovodenje« omogoča operacije kot npr. zaustavitev opravila, ki zahtevajo sprostitev vseh virov opravila. Da bi bila koda čim manjša, vsi ti seznami uporabljajo iste funkcije za dodajanje, brisanje in preiskovanje seznama.



Slika 3. Primer sistemskega seznama elementov različnih velikosti.

Vsak element seznama se začne s strukturo ***list_header_s***, ki vsebuje kazalec na naslednji element ***next*** in kazalec na lastnika elementa ***owner***. Slednji je običajno[3] kazalec na proces, ki je element ustvaril in omogoča čiščenje[4] po koncu izvajanja procesa.

Funkcije za dodajanje, brisanje in preiskovanje seznama v resnici zanima le struktura ***list_header_s*** (točneje ... berejo in spreminjajo le njeno polje ***next***). Podatki, ki ji sledijo, so za njihovo delovanje nebistveni. Tako lahko iz katerekoli podatkovne strukture naredimo člana seznama – če ji le na začetek dodamo ***list_header_s***.

[3] Obstajata dve izjemi: vrednosti (***0x0000***) KERNEL in (***0xffff***) USER. Prva pove, da je lastnik strukture operacijski sistem, druga pa se uporablja pri uporabniški kopici za označevanje prostih pomnilniških blokov brez lastnika.

[4] Ko proces konča delo, se operacijski sistem sprehodi skozi sezname vseh sistemskih virov in »počisti« za njim.

Naslednja koda prikazuje to strukturo.

```
/* each linked list must start with list_header */
typedef struct list_header_s {
        void *next;
        void* owner; /* pointer to task or 0 for kernel owned objects */
} list_header_t;
```

Poglavje 5 Upravljanje s pomnilnikom

Upravljanje s pomnilnikom je večnivojsko in kompleksno področje. Denimo, zgolj funkcija standardne knjižnice C ***malloc()*** je dolga 5.289 vrstic izvorne kode v jeziku C, čeprav deluje le v linearnem naslovnem prostoru in ne vsebuje izvedbe navideznega pomnilnika, ki je implementirana na nižji ravni.

Mavrica ne premore ne nadzorniškega procesorskega načina, ne enote za upravljanje s pomnilnikom.

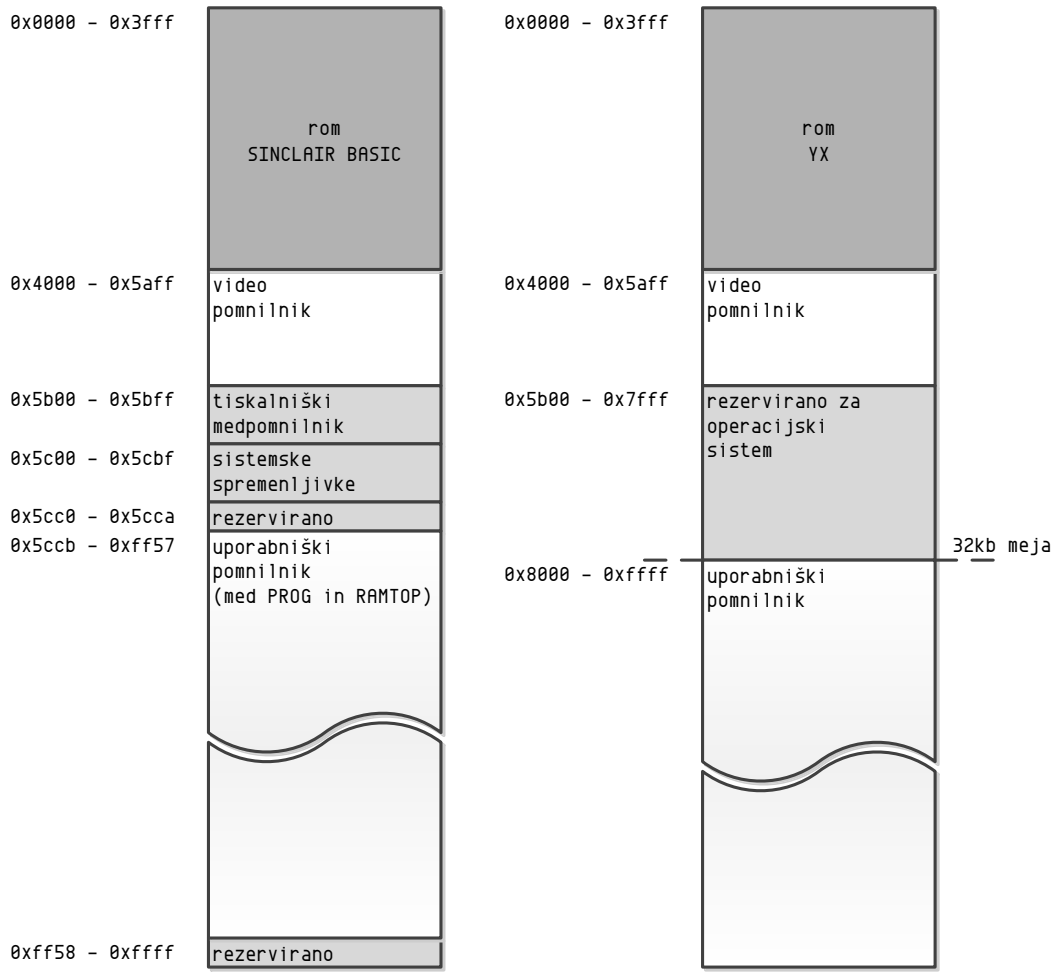
> *Teoretično bi bila razširitev naslovnega prostora s preklapljanjem na diskovno ali drugo počasnejšo pomnilniško enoto in njegova izolacija vsaj delno možna.*
>
> *32 kilobajtovrazpoložljivega pomnilnika bi lahko razdelili v dvokilobajtne pomnilniške bloke. Med meta podatke vsakega opravila bi dodali 16-bitno besedo, vsak bit bi pomenil rezervacijo enega izmed šestnajstih dvokilobajtnih blokov. Ob preklopu opravila bi se primerjalo rezervirane pomnilniške bloke starega in novega opravila in ugotovilo, katere pomnilniške bloke je treba popolniti z diska in katere shraniti na disk.*
>
> *Vendar je to izvedljivo le s sodobno strojno opremo. Strojna oprema, ki je obstajala v osemdesetih in je združljiva z Mavrico, namreč ni dovolj hitra za (resno) izvedbo navideznega pomnilnika.*

5.1 Organizacija pomnilnika

Procesor Z80 lahko naslovi 64 kilobajtov pomnilnika. Pri Mavrici je v prvih 16 kilobajtov naslovnega prostora preslikan bralni pomnilnik, v katerem je izvirno shranjen Sinclair Basic, v različici z YX pa operacijski sistem. Naslednjih 6.912 bajtov je video pomnilnik.

Pri različici s Sinclair Basicom sledijo tiskalniški izravnalnik, prostor za sistemske spremenljivke in rezerviran prostor, sam uporabniški pomnilnik pa je med sistemskima spremenljivkama PROG in RAMTOP.

Pri različici z YX pa naslovni prostor od konca video pomnilnika do 32-kilobajtne meje operacijski sistem uporabi za svoj sklad, sistemske spremenljivke in kopico, uporabniški pomnilnik pa se začne nad mejo 32 kilobajtov.



Slika 4. Organizacija pomnilnika: Sinclair Basic in jedro YX.

Algoritmi upravljanja s kopico imajo različne lastnosti. Nekateri zasledujejo cilj optimalne hitrosti rezerviranja pomnilniških blokov, drugi njihovega sproščanja, tretji najbolje izkoriščajo razpoložljivi prostor itn. [5]

Pri funkcijah za upravljanje s kopico operacijskega sistema YX sta bila glavna cilja:

- da iste funkcije upravljajo kopico operacijskega sistema in uporabniško kopico ter po potrebi tudi nelinearno kopico[5] in
- da je koda funkcij čim krajša.

Kopico operacijskega sistema najdemo med naslovoma ***0x5b00*** in ***0x7fff***. Točna lokacija in velikost sta odvisni od različice operacijskega sistema. Ne glede na različico pa se razteza vse do naslova ***0x7fff***.

Naslov kopice določi zagonska datoteka crt0.s v zadnjih dveh vrsticah (segment _HEAP).

Uporabniška kopica se začne na naslovu ***0x8000*** in se razteza do naslova ***0xffff***, je velikosti 32 kilobajtov.

Ključni funkciji za upravljanje s kopico sta ***mem_allocate()*** in ***mem_free()***. Obe kot prvi parameter prejmeta kazalec na kopico, s čimer je izpolnjen cilj, da lahko iste funkcije upravljajo več različnih kopic.

5.1.1 Upravljanje kopice

Kopica je organizirana kot seznam pomnilniških blokov. Na začetku vsakega bloka je struktura ***block_t***.

```
/* block status, use as bit operations */
#define NEW             0x00
#define ALLOCATED       0x01

typedef struct block_s {
        list_header_t   hdr;
        byte            stat;
        word            size;
        byte            data[1];
} block_t;

extern void *heap;

extern void mem_init(void *heap, word size);
extern void *mem_allocate(void *heap, word size, void *owner);
extern void *mem_free(void *heap, void *p);
extern void mem_copy(byte *src, byte *dest, word count) __naked;
```

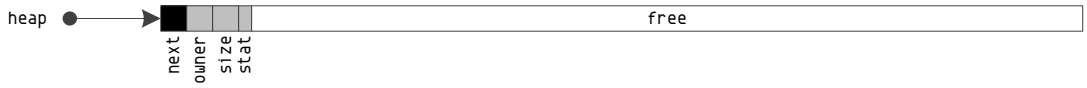
[5] Kopico, ki ni monoton pomnilniški blok, ampak jo sestavlja več pomnilniških blokov, povezanih v seznam.

Struktura se začne z glavo ***hdr***, ki vsebuje kazalec na naslednji element ***next*** in podatek o lastniku bloka ***owner***. Polje ***stat*** pove, ali je blok rezerviran ali prost. Velikost bloka pa določa polje ***size***.

Uporabniku so podatkovne strukture kopice nevidne. Vedno operira s kazalcem na podatkovni blok ***data***. Za ustrezno upoštevanje glave bloka in prištevanje in odštevanje dodatnih potrebnih bajtov skrbi jedro.

5.1.1.1 Inicializacija kopice

Da bi lahko kopico začeli uporabljati, jo moramo pripraviti. Inicializirana kopica je prvi pomnilniški blok v seznamu. Označen mora biti za prostega, njegova velikost mora biti velikost kopice, zmanjšana za velikost strukture ***block_t***. Blok nima naslednikov, njegov lastnik pa je operacijski sistem.



Slika 5. Inicializirana kopica.

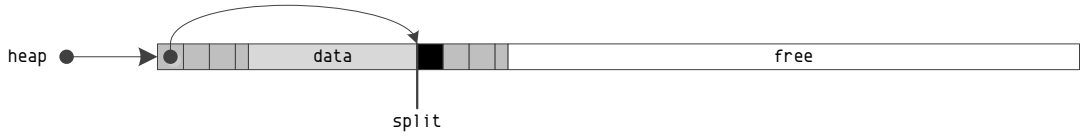
Kopico jedra in uporabniško kopico skladno s pomnilniško sliko inicializira jedro ob zagonu. Tule je prvih pet vrstic standardne funkcije ***main()***.

```
        /*
         * initialize operating system and user heap
         */
        mem_init(&sys_heap, SYS_HEAP_END - &sys_heap + 1);
        mem_init((void*)USR_HEAP_START, USR_HEAP_END - USR_HEAP_START + 1);
```

Ustrezne konstante in funkcije so definirane v datotekah system.c in system.h.

5.1.1.2 Rezervacija pomnilniškega bloka

Pomnilniški blok rezerviramo s funkcijo ***mem_allocate()***. Funkcija v seznamu vseh pomnilniških blokov – vključno s kopico, ki je zgolj eden izmed njih – išče ustrezno velik pomnilniški blok, ki ni zaseden.

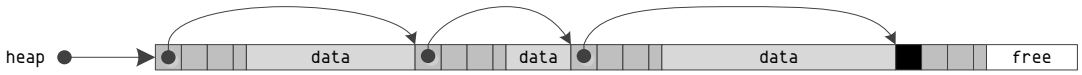


Slika 6. Rezerviran pomnilniški blok in preostanek kopice.

Ko ga najde, sprejme odločitev o smiselnosti razdelitve na dva bloka: enega velikosti, ki je zahtevana z ***mem_allocate()***, in na ostanek. Če je ostanek večji od štirih bajtov, se razdelitev bloka izvede, v nasprotnem primeru pa se uporabi cel blok. Prvi blok se označi za zasedenega, morebitni novonastali pa za prostega.

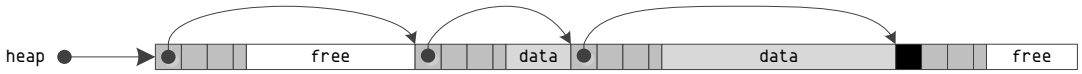
5.1.1.3 Sprostitev pomnilniškega bloka

Pomnilniški blok sprostimo s funkcijo ***mem_free()***. Ta v polje ***stat*** bloka zapiše vrednost ***NEW*** in ga s tem razglasi za prostega. Da bi zagotovili optimalno delovanje kopice, se po tej operaciji preveri še prejšnji in naslednji blok. Če je kateri od njiju tudi prost, se vse proste bloke zlije. Na ta način se ustvari nov dolg, neprekinjen pomnilniški blok.



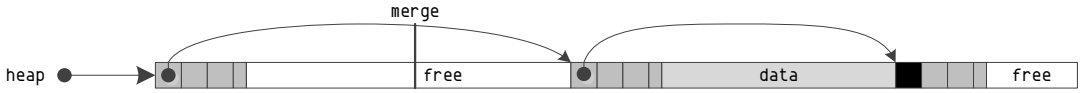
Slika 7. Trije pomnilniški bloki in kopica (kot zadnji blok).

Zgornja slika prikazuje tri pomnilniške bloke. Ob sproščanju prvega bloka nastane novo stanje.



Slika 8. Stanje po sprostitvi prvega bloka.

Ko sprostimo še drugi blok, se prvi in drugi blok zlijeta v en neprekinjen blok.



Slika 9. Stanje po sprostitvi drugega bloka, prvi blok in drugi blok se zlijeta.

Če sprostimo vse rezervirane bloke, na koncu ostane le en blok. Ta blok ni nič drugega kot začetna kopica.

Poglavje 6 Časovni razporejevalnik

Vsako petdesetinko sekunde Mavrica osveži zaslon. Ob tem se sproži prekinitev. V prekinitvenem načinu IM1 se ob nastopu prekinitve trenutni naslov (prekinjenega programa, ki se je takrat izvajal) najprej shrani na sklad in potem izvrši skok na naslov ***0x38***. Mavrica v tem načinu vse druge prekinitve onemogoči, dokler jih z ukazom EI znova ne omogočimo. Iz prekinitve se vrnemo z ukazom ***RETI***, ki s sklada pobere naslov prekinjenega programa in skoči nanj.

Program na naslovu ***0x38*** na sklad shrani vse registre in alternativne registre, da bi jih lahko pred izhodom iz prekinitve vrnil v prvotno stanje in prekinjenemu programu omogočil nemoteno izvajanje. Potem pa drugega za drugim pokliče:

- Časovnike in
- časovni razporejevalnik.

Naloga zadnjega je, da razporeja čas med opravila jedra.

6.1 Časovniki

Jedro vzdržuje seznam časovnikov. Vsak element seznama vsebuje kazalec na funkcijo *hook*, ki naj se sproži ob nastopu pogojev časovnika, in zaželeni časovni interval (v petdesetinkah sekunde) proženja ***ticks***. Strukturo elementa seznama prikazuje naslednja koda.

```
/*
 * timer ticks 50 times per second
 */
typedef struct timer_s {
        /* list_header_t compatible header */
        list_header_t hdr;

        void (*hook)();         /* hook routine */
        word ticks;             /* trigger after ticks */
        word _tick_count;       /* count ticks (internal) */
} timer_t;
```

Med vsako prekinitvijo se potem pokliče funkcija ***tmr_chain()***.

```
/*
 * chain timers
 * note:        this is done inside 50 hz interrupt
 *              so no di/ei calls are needed.
 */
void tmr_chain() {
        timer_t *t=tmr_first;
        while(t) {
                if (t->_tick_count==0) { /* trig it */
                        t->_tick_count=t->ticks;
                        t->hook();
                } else t->_tick_count--;
                t=t->hdr.next;
        }
}
```

Ta funkcija se sprehodi skozi vse časovnike, in če je interni števec časovnika ***tick_count*** dosegel zaželeni časovni interval, ***ticks*** resetira interni števec časovnika in sproži funkcijo ***hook()***.

Za dodajanje in odvzemanje časovnika iz seznama časovnikov skrbita funkciji ***tmr_install()*** in ***tmr_uninstall()***.

```
extern timer_t *tmr_install(void (*hook)(), word ticks, void *owner);
extern timer_t *tmr_uninstall(timer_t *t);
```

Gonilniki naprav, ki morajo periodično preverjati stanje strojne opreme in ob registraciji gonilnika to sporočijo operacijskemu sistemu, so izvedeni s časovniki. Tako, denimo, deluje gonilnik za tipkovnico, ki je opisan v posebnem poglavju.

6.2 Časovni razporejevalnik

6.2.1 Opravilo

Časovni razporejevalnik razporeja procesorski čas med opravila. Uporabljena metoda razporejanja je round robin *[6]*.

```
typedef struct task_s {
        /* list_header_t compatible header */
        list_header_t hdr;
        /* task properties */
        word sp;                /* stack pointer. task context is stored there. */
        event_t **wait;         /* event list or null */
        byte num_events;        /* number of events in event list */
        byte state;             /* task state (bits 0-1), bits 2-7 are reserved */
} task_t;
```

Opravilo je najmanjša enota izvajanja, analogija v sodobnih jedrih je nit (ang. thread). Tako kot vse druge sistemske vire, se tudi vsa opravila vodi v seznamu. Vsako opravilo ima svoj sklad. Polje ***state*** je stanje opravila in pove, ali opravilo v danem trenutku teče ali je zaustavljeno. Če je zaustavljeno, kombinacija polj ***num_events*** in ***wait*** pove, na katere dogodke opravilo čaka.

6.2.2 Kazalec na opravilo kot lastnik sistemskega vira

Vsak sistemski vir (na primer: pomnilniški blok, časovnik ali dogodek) ima lastnika. Oznako lastnika določa polje ***owner*** in je del strukture ***list_header_t***.

Kadar lastnik sistemskega vira ni operacijski sistem, se za v polje ***owner*** vpiše kar kazalec na opravilo ***task_t ****. Tako lahko jedro po koncu izvajanja opravila preišče vse njegove sistemske vire in jih sprosti.

Opravilo je lastnik lastnih virov, ne pa tudi samega sebe. Lastnik opravila je operacijski sistem.

6.2.3 Dodajanje novega opravila

Opravila ustvarimo s klicem funkcije ***tsk_create()***. Le-ta prejme dva parametra. Prvi je vstopna točka opravila (tj. kazalec na funkcijo opravila), drugi pa velikost sklada. Funkcija ***tsk_create()*** na ustrezni kopici rezervira pomnilniški blok in kazalec nanj zapiše v člana ***sp*** strukture ***task_t***.

Delovanje funkcije je najbolje razloži izvorna koda.

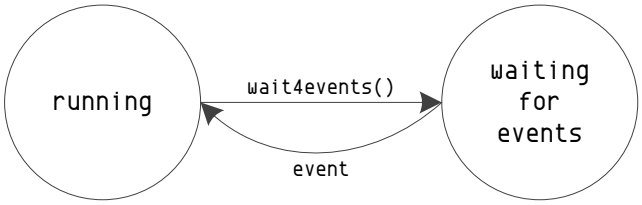
```
/*
 * create task.
 * 1) allocate slot,
 * 2) allocate stack,
 * 3) create empty context and put it on stack incl. entry_point as return address,
 * 4) store sp,
 * 5) set state to running.
 */
task_t * tsk_create(void (*entry_point)(), uint16_t stack_size) {

        task_t *t;
        void *stack;
        word *ret_addr;

        if ( t = (task_t *)syslist_add((void **)&tsk_first_running, sizeof(task_t),
SYS) ) {
                /* allocate stack for the task */
                stack=mem_allocate((void*)USR_HEAP_START, stack_size, (void *)t);
                if (!stack) {
                        /* was allocated - so free it */
                        syslist_delete((void **)&tsk_first_running, (void *)t);
                        t=NULL;
                } else {
                        t->wait=NULL;
                        t->state=TASK_STATE_RUNNING;
                        /* prepare stack */
                        t->sp=(word)stack + stack_size - CONTEXT_SIZE;
                        /* top two bytes are the return address */
                        ret_addr=(word *)(t->sp + CONTEXT_SIZE - 2);
                        (*ret_addr)=(word)entry_point;
                }
        }
        return t;
}
```

Funkcija jedra najprej poskrbi, da so vsa druga opravila ustavljena in da med dodajanjem opravila ne bi prišlo do preklopa opravil. Nato rezervira pomnilniški blok za opravilo. Zanj alocira sklad in shrani kazalec na sklad opravila. Le-tega nastavi na vrh sklada minus kontekst opravila na skladu (tj. shranjeni registri opravila in naslov za vrnitev). Stanje opravila nastavi

na »running«. In na koncu na sklad zapiše vstopno točko v opravilo. Ob naslednjem preklopu bo jedro avtomatično pobralo s sklada vrednosti vseh registrov in skočilo na vstopno točko.

6.2.4 Stanje opravila in dogodki

V sodobnem jedru ima lahko opravilo, tj. nit, precej različnih stanj. Jedro YX ima le dve vrsti. Opravilo je lahko le v vrsti aktivnih opravil ali v vrsti čakajočih (na dogodke).



Slika 10. Edini možni stanji opravila.

Dogodki omogočajo optimalnejšo izrabo procesorskega časa. Dogodek je lahko le v enem izmed dveh stanj: signaled ali non-signaled.

```
typedef enum event_state_e {
        nonsignaled,
        signaled
} event_state_t;

typedef struct event_s {
        /* list_header_t compatible header */
        list_header_t hdr;
        /* event state */
        event_state_t state;
} event_t;
```

Če opravilo ne želi po nepotrebnem tratiti procesorskega časa, lahko pokliče funkcijo ***wait4events()*** in z njo jedru naroči, naj ga časovni razporejevalnik pri preklopu opravil ignorira tako dolgo, dokler se ne zgodijo dogodki, na katere čaka.

Šolski primer je, denimo, čakanje na rezultat vhodno-izhodne operacije. Na primer čakanje na pritisk tipke. Jedro tako ali tako vsako petdesetinko sekunde preveri stanje tipkovnice. Nobene potrebe ni, da bi opravilo, ki le čaka na nov dogodek v tej vrsti, tratilo procesorski čas. Bolj smiselno ga je zamrzniti in ga odmrzniti šele, ko koda za stanje tipkovnice oznani, da je na voljo nov dogodek.

Funkcija, ki opravilo postavi v vrsto čakajočih, ni posebej zahtevna.

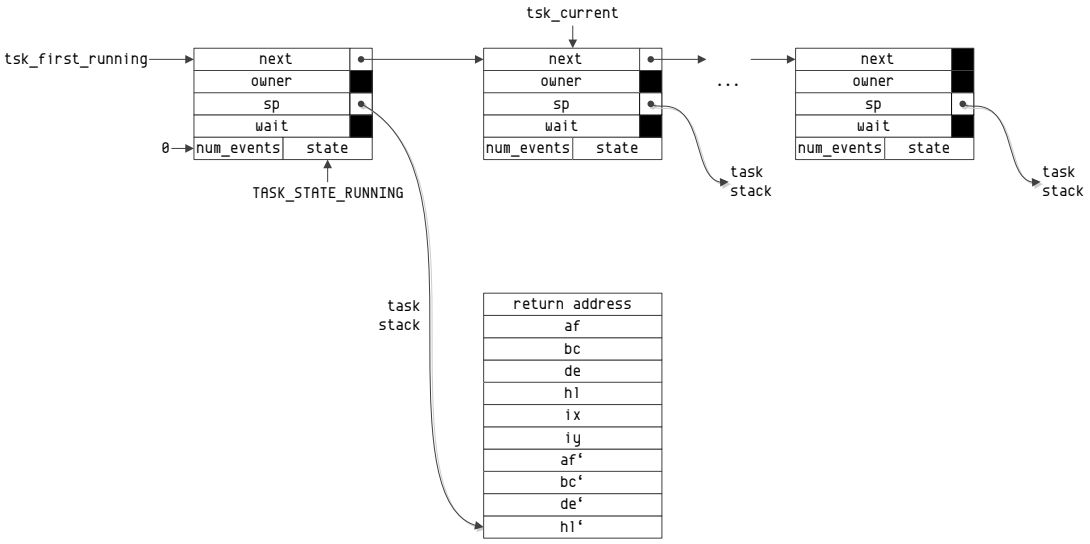
```
void tsk_wait4events(event_t **e, byte num_events) {

        __asm
                di                      /* must disable */
        __endasm;

        /* set task flags to waiting */
        tsk_current->wait=e;
        tsk_current->num_events=num_events;
        tsk_current->state = TASK_STATE_WAITING;

        /* move task from running to waiting list... */
        list_remove(&tsk_first_running, tsk_current);
        list_insert(&tsk_first_waiting, tsk_current);

        /* ...and switch */
        tsk_switch();
}
```

Najprej onemogoči prekinitve. Nato opravilo premakne iz vrste aktivnih v vrsto čakajočih oz. spečih. In na koncu izvede preklop opravila. Del preklopa opravila pa je tudi sproščanje vseh opravil, katerih dogodki so na voljo.

```
/* release all waiting tasks */
curr=tsk_first_waiting;
while (curr) {
        t=curr;
        curr=curr->hdr.next;
        if (has_signaled_events(t)) {
                /* move task from running to waiting list... */
                t->state = TASK_STATE_RUNNING;
                list_remove(&tsk_first_waiting, t);
                list_insert(&tsk_first_running, t);
        }
}
```

6.2.5 Preklop opravila

Tudi opravilo je zgolj vir operacijskega sistema. Kot tako je shranjeno v seznamu vseh opravil oz. v eni izmed obeh vrst: vrsti opravil, ki tečejo, in opravil na čakanju. Blok opravila vsebuje osnovne podatke o opravilu, ne pa tudi konteksta opravila (tj. vseh registrov opravila in naslova za skok), saj je ta shranjen na lokalnem skladu opravila.



Slika 11. Vrsta opravil v izvajanju.

Ob preklopu opravila se najprej na sklad trenutnega opravila shranijo vsi registri in naslov, kjer je bilo opravilo prekinjeno. Potem se izvede funkcija ***tmr_chain*()**, ki sproži vse registrirane časovnike. Nato se požene funkcija ***select_next_task()***, ki se z uporabo preprostega round robin algoritma in s preverjanjem dogodkov, ki so se vmes sprožili, odloči, katero bo naslednje opravilo. In na koncu se spremeni kazalec s trenutnega opravila na novo opravilo. Od tam prebere kazalec na sklad. S sklada pobere vse shranjene registre. Omogoči prekinitve in izvede ukaz ***RETI***, ki skoči na naslov, shranjen na skladu (oz. naslov, kjer je bilo opravilo prekinjeno ob zadnjem preklopu).

Kadar naslednjega opravila ni, se sproži podprogram operacijskega sistema ***tar_pit()***, ki poskrbi, da ostane operacijski sistem še naprej aktiven.

Poglavje 7 Sistemski klici

V sedanji izvedbi jedro uporabimo tako, da svoj program (z več opravili) zgradimo skupaj z njegovo izvorno kodo in oboje zapišemo v ROM. Zato je mogoče funkcije jedra klicati kar neposredno. Kljub temu pa so sistemski klici ustrezno protokolirani, saj je s tem vzpostavljena osnova za nadgradnjo jedra v operacijski sistem.

Za sistemski klic je uporabljen ukaz ***RST 0x10***. Pred tem moramo na sklad shraniti kazalec na ime strežnika.

```
                ld      hl,#_yx
                push    hl
                rst     0x10
                ;; hl now points to of functions of server yx
_yx:
                .asciiz "yx"
```

Po klicu je v registru ***HL*** kazalec na tabelo funkcij strežnika z imenom »yx«, tj. strežnika jedra. Za lažje delo obstaja funkcija ***void *query_interface(string api)***, ki vrne kazalec na tabelo funkcij strežnika. V jeziku C funkcije strežnika YX kličemo takole:

```
#include "yx.h"

yx_t *yx;

void main() {
        yx=(yx_t *)query_interface("yx");
        void *kilobyte_block=yx->malloc(1024);
        yx->free(kilobyte);
}
```

Znotraj funkcij YX je jedru na voljo globalna spremenljivka ***current_task***, ki kaže na trenutno opravilo, tj. kdo je funkcijo poklical, kar omogoča uporabo ustreznih sinhronizacijskih mehanizmov tam, kjer je to potrebno.

7.1 Strežniki

V prejšnjem poglavju smo spoznali strežnik »YX«. Ta pa ni nujno edini, saj jedro omogoča registracijonovih strežnikov in vsak strežnik ima lastno tabelo funkcij. Tule je del tabele funkcij strežnika YX, tj. jedra.

```
struct yx_s {
        /* memory management */
        void* (*malloc)(word size);
        void (*free)(void *p);
        void (*memcpy)(byte *src, byte *dst, word count);
        /* ... */
}
```

Vizija jedra je, da novi strežniki, ki se lahko registrirajo tudi kot programi in niso nujno del jedra, dodajajo funkcionalnosti, ki jih jedro ne nudi. Vse, kar mora strežnik storiti za registracijo, je:

1) pripraviti svojo tabelo funkcij,

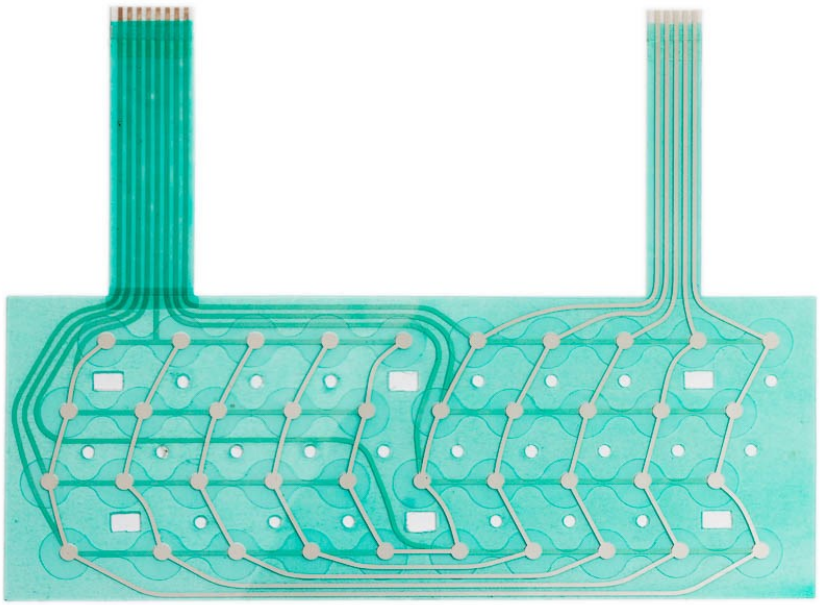
2) jo registrirati s klicem funkcije strežnika YX ***void register_interface(char *name, void *fntbl)***, tako postane s funkcijo ***query_interface()***, tabela dostopna vsem drugim opravilom.

3) pripraviti čelno datoteko C s prototipi funkcij.

Na ta način lahko, denimo, strežnik za grafični uporabniški vmesnik doda funkcije za delo z okni in sporočili; strežnik za internet funkcije za delo z vtičniki (ang. sockets) itn.

Poglavje 8 Tipkovnica

Del diplomske naloge je bil napisati sodoben krmilnik za mavričino tipkovnico. Le-ta je preprosta mreža stikal z naslovnim vodilom in podatkovnim vodilom, kot prikazuje naslednja slika.



Slika 12. Mavričina tipkovnica, vir: internet, licenca Creative Commons.

Stikala so povezana z dvema vodiloma: naslovnim vodilom in podatkovnim vodilom. Ob pritisku na tipko in ustrezni vsebini naslovnega vodila se sklene tokokrog in podatki se preslikajo na podatkovno vodilo.

> *Da je mavričina tipkovnica tako preprosta, da bi jo lahko narisali na dva lista papirja z grafitnim svinčnikom, ima svoje omejitve. Denimo, če pritisnemo tri tipke hkrati, sta dve izmed njih lahko v isti vrsti in ju je po izhodu nemogoče ločiti. Ta pojav je bil v starih in cenejših tipkovnicah pogost in se imenuje ghosting.*

8.1 Branje tipkovnice

Branje tipkovnice poteka preko izhodnih vrat. Naslednja tabela prikazuje povezave tipk in naslovov vrat ter številko bita, ki je ugasnjen, če je stikalo sklenjeno[6], in prižgan, če ni.

PORT	**BIT**				
	0	**1**	**2**	**3**	**4**
0XF7FE	5	4	3	2	1
0XEFFE	6	7	8	9	0
0XDFFE	Y	U	I	O	P
0XBFFE	H	J	K	L	<ENTER>
0X7FFE	B	N	M	<SYMBOL>	<SPACE>
0XFEFE	V	C	X	Z	<SHIFT>
0XFDFE	G	F	D	S	A
0XFBFE	T	R	E	W	Q

Če bi torej ob branju vsebine vrat 0xf7fe zaznali, da je bit 4 ugasnjen, bi to pomenilo, da je stikalo za tipko 1 sklenjeno.

> *Naslov vrat je sestavljen iz manj pomembnega bajta, ki je vedno 0xfe, in bolj pomembnega bajta, ki je negirana naslovna linija, ki jo želimo brati. Zato je en bit bolj pomembnega bajta vedno ugasnjen (ta bit je v resnici naslov, ki ga beremo).*

Jedro YX preveri stanje tipkovnice petdesetkrat na sekundo. Pri dekodiranju pritisnjenih tipk je pomembna tudi časovna komponenta. Dejstvo, da je stikalo sklenjeno, ne pomeni nujno, da je uporabnik pravkar pritisnil tipko. Morda je od zadnjega branja preprosto še ni izpustil? Ob branju tipkovnice petdesetkrat na sekundo je to zelo pogosto.

Da bi dekodirali, kaj se je zgodilo, smo razvili inovativen algoritem, ki zazna spremenjene tipke s preprostimi logičnimi operacijami. Ob nastopu prekinitve jedro eno za drugo prebere vse naslovne linije. Zaradi majhnega števila tipk (Mavrica jih ima le 40) lahko stanje vsake tipke spravi v svoj bit in celotno stanje tipkovnice zasede le 5 bajtov. Ko je branje končano, prebranih pet bajtov s kombinacijo logičnih operacij združi s petimi bajti iz prejšnjega branja in na ta način ugotovi, katere tipke so bile od zadnjega branja pritisnjene in katere izpuščene.

[6] Vrednost 0 **in ne 1** pomeni sklenjeno stikalo.

	bit	39		5	4	3	2	1	0
A	Staro stanje	0	...	1	1	0	0	1	0
B	Novo stanje	0	...	1	0	0	1	1	0
C	Spremembe (A XOR B)	0	...	0	1	0	1	0	0
D	Pritisnjene (A AND C)	0	...	0	1	0	0	0	0
E	Spuščene (B AND C)	0	...	0	0	0	1	0	0

Slika 13. Dekodiranje pritisnjenih in izpuščenih tipk z logičnimi operacijami.

Na zgornji sliki s pomočjo stanja tipkovnice iz prejšnjega branja in sveže prebranega stanja tipkovnice s preprostimi logičnimi operacijami iz petih bajtov (40 bitov, vsak predstavlja stanje stikala) ugotovimo spremembe. Zaporedna številka bita je kazalec v dekodirno tabelo tipk.

8.2 Dekodiranje tipk

V prejšnjem poglavju opisani postopek branja tipkovnice v prekinitvi in ugotavljanje, katere tipke so bile pritisnjene in spuščene, je vhod v dekodiranje tipke. Kot je bilo že omenjeno, je zaporedna številka bita pritisnjene ali izpuščene tipke kazalec v dekodirno tabelo. Primer takšne tabele.

```
_kbd_map:
                .byte '5',    '4',    '3',    '2',    '1'
                .byte '6',    '7',    '8',    '9',    '0'
                .byte 'y',    'u',    'i',    'o',    'p'
                .byte 'h',    'j',    'k',    'l',    0x0d
                .byte 'b',    'n',    'm',     0x01, 0x20
                .byte 'v',    'c',    'x',    'z',    0x02
                .byte 'g',    'f',    'd',    's',    'a'
                .byte 't',    'r',    'e',    'w',    'q'
```

S tem pa dekodiranja še ni konec. Ker je bil eden izmed ciljev krmilnika tipkovnice, da je uporaben tako za konzolne aplikacije kot za grafične uporabniške vmesnike, se dogodki o pritisnjenih in izpuščenih tipkah shranijo v 32 dogodkov dolg krožni seznam. To omogoča, da uporabnik pritiska tipke tudi takrat, ko jedro počne kaj drugega in so obvestila o teh dogodkih na voljo pozneje.

Zadnji korak pri dekodiranju tipk je iz zaporedja posamičnih dogodkov tipkovnice ugotoviti, kdaj je bila kakšna tipka pritisnjena, kdaj izpuščena in ali je bila pred njo morda pritisnjena tipka Shift ali tipka Symbol, ter s tem spremeniti interpretacijo.

To dekodiranje je sicer prepuščeno konzoli oz. grafičnemu uporabniškemu vmesniku in ni del jedra, vendar je bilo razvito v enem primeru uporabe jedra.

```
char shell_get_char() {
        static byte *map=(byte *)&kbd_map; /* current map */
        byte key;
        char ch;

        if (key=kbd_read()) {
                ch=map[key&0b00111111]; /* get char */
                if (key&KEY_DOWN_BIT) { /* key down */
                        switch(ch) {
                                case 0x01: /* symbol */
                                        map=(byte *)&kbd_map_symbol;
                                        break;
                                case 0x02: /* shift */
                                        map=(byte *)&kbd_map_shift;
                                        break;
                        }
                        return 0; /* no key down shall be propagated */
                } else { /* key up */
                        switch(ch) {
                                case 0x01: /* symbol */
                                case 0x02: /* shift */
                                        map=(byte *)&kbd_map;
                                        return 0;
                                default:
                                        return ch;
                        }
                }
        } else return 0; /* no keys */
}
```

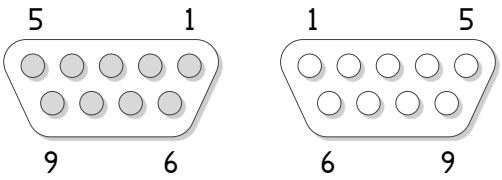
Koda deluje tako, da ob pritisku na tipko Symbol ali Shift zamenja dekodirno tabelo za tipkovnico. Ko je tipka izpuščena, pa ponovno vzpostavi osnovno dekodirno tabelo.

Poglavje 9 Zaporedna vrata

Mavrica z dodatkom Interface 1 vsebuje zaporedna vrata RS-232, ne pa tudi krmilnika zanje. Zato jih je treba krmiliti programsko, z metodo bitne eksplozije [8].

9.1 Priključek

Ker ima mavričin priključek RS-232 nekoliko drugačno razporeditev nožic od običajnega (standardnega), običajni kabel RS-232 ne bo deloval. Spodnja slika kaže razporeditev nožic na osebnem računalniku in Mavrici.



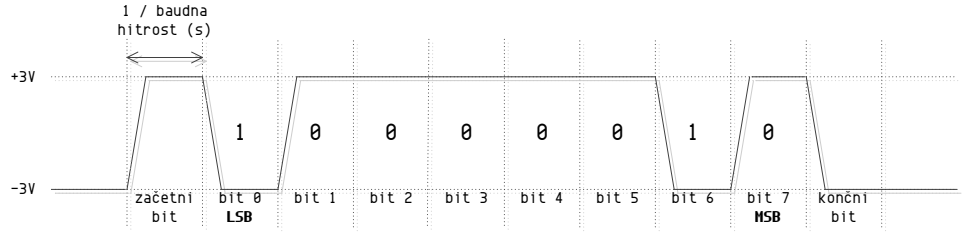
Priključek v osebni računalnik	**Priključek v Mavrico**
2-RD	3-TD
3-TD	2-RD
5-GND	7-GND
7-RTS	4-CTS
8-CTS	5-RTS
1-DCD	9-DTR

9.2 Pošiljanje podatkov po zaporednih vratih RS-232

Biti se po zaporednih vratih RS-232 pošiljajo zaporedno v časovnih razmikih. Časovni razmik med dvema bitoma določa hitrost prenosa v baudih, hitrost se izračuna kot 1 / baudna hitrost. Običajne (bolj znane) hitrosti prenosov so 2.400 baudov, 9.600 baudov, 19.200 baudov, 28.800 baudov, 57.600 baudov in 115.200 baudov. Pri 2.400 baudih se torej en bit prenese v 1/2.400 sekunde.

Poleg podatkovnih bitov se čez zaporedna vrata prenašajo tudi kontrolni biti. V konfiguraciji 2400-8-N-1, kar je okrajšano za: hitrost 2.400 baudov, velikost podatka 8 bitov (=1 bajt), prenaša se brez paritete in z enim zaustavitvenim bitom, se za vsak preneseni bajt podatkov prenese 10 bitov (8 bitov in 2 kontrolna bita za en bajt). To pomeni, da je 2.400 baudov v tej konfiguraciji enako hitrosti 240 bajtov na sekundo, če ne upoštevamo morebitnih zakasnitev zaradi RTS/CTS ali kakšnega drugega protokola.

Ko na liniji ni podatkov, je napetost na njej -3V. Ta napetost predstavlja logično 1. Prenos se začne z začetnim bitom (logična 0) in konča z zaustavitvenim bitom (logična 1). Naslednja slika prikazuje pošiljanje enega znaka, tj. črke 'A' = $41_{(16)}$ = $01000001_{(2)}$, po zaporednih vratih v načinu 8-N-1:



Slika 14. Menjavanje signala pri zaporednem prenosu črke 'A'.

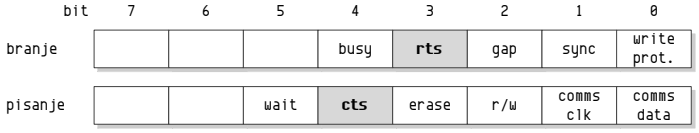
Način 8-N-1 pomeni, da pošiljamo 8 bitov/znak, brez paritete in z 1 končnim bitom.

Da bi črko lahko poslali, moramo spreminjati napetost na ustrezni nožici (TD) vrat, pri čemer je treba upoštevati, da sprememba napetosti na vratih potrebuje nekaj časa (tj. poševne prehode signalov).

9.3 Programiranje zaporednih vrat RS-232

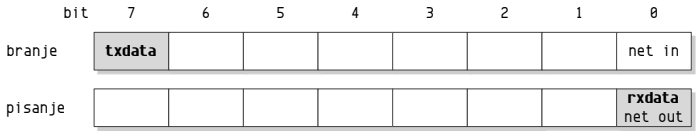
Za programiranje uporabljamo dvoje vrat: ***0xef*** in ***0xf7***. Stanje na nožicah priključka RS-232 programiramo z uporabo strojnih ukazov ***IN*** in ***OUT***.

0xef uporabljamo za branje signala RTS (ready to send) in za manipulacijo signala CTS (clear to send),



Slika 15. Biti vrat 0xef.

0xf7 pa za sprejem in pošiljanje podatkovnih bitov.



Slika 16. Biti vrat 0xf7.

Ker se oboje vrat uporablja tudi v druge namene (za programiranje enote microdrive in lokalne računalniške mreže), so biti, relevantni za programiranje zaporednih vrat RS-232 označeni z modro barvo.

Vrednosti na vratih so negirane. Vrednost bita 0 na vratih pomeni logično 1 (-3V) na nožici priključka in vrednost bita 1 na vratih pomeni logično 0 (+3V) na nožici priključka.

9.3.1 Kako hitro lahko pošilja in sprejema Mavrica?

Uradno je največja baudna hitrost z vmesnikom Interface 1 19.200. Teoretično pa so hitrosti lahko veliko večje. Mavričin procesor deluje na 3.5 MHz, torej 3,500.000 urnih period na sekundo. Vsak strojni ukaz traja določeno število urnih period (najmanj 4).

Ob upoštevanju dejstva, da ukazi za branje in pisanje enega bita na zaporedna vrata trajajo najmanj 30 urnih period, lahko izračunamo maksimalno teoretično hitrost prejemanja in pošiljanja podatkov:

$$\frac{3.500.000}{30} = 116.666\ baudov$$

To pomeni, da lahko teoretično podatke v Mavrico in iz nje prenašamo s standardno hitrostjo 115.200 baudov oziroma **šestkrat hitreje od uradno največje hitrosti**.

V praksi se izkaže, da zaradi omejenega nabora Z80-ukazov podatke lahko zanesljivo izmenjujemo s hitrostjo 57.600 baudov in relativno zanesljivo s hitrostjo 115.200 baudov. Pri slednji hitrosti je zanesljiv prenos možen le z uporabo protokola, ki zaznava in odpravlja napake.

9.3.2 Protokol RTS/CTS

Da bi se lahko različno prepustne naprave – npr. osebni računalnik in modem – pogovarjale med sabo, protokol RS-232 predpisuje nekaj možnih načinov usklajevanja naprav. YX uporablja usklajevanje RTS/CTS. To 30 let stari Mavrici omogoča, da računalniku s pomočjo signala CTS sporoči, kadar ni več sposobna sprejemati podatkov. Osebni računalnik čaka toliko časa, da Mavrica z aktivacijo signala CTS (clear to send) sporoči, da je znova sposobna sprejemati podatke.

Novejše izvedbe protokola RS-232 pri komunikaciji med dvema računalnikoma (DTE z DTE) uporabljajo signal RTS prav tako kot CTS. Teoretično bi torej Mavrica morala pred pošiljanjem bita preveriti, ali ga je računalnik sposoben sprejeti. V praksi pa so sodobni računalniki toliko hitrejši, da do tega nikoli ne pride.

Protokol RTS/CTS predpisuje, da lahko tudi ko računalnik s CTS sporoči, da ne more več sprejemati podatkov, pride še nekaj bajtov. Starejši računalniki so zaradi nizkih hitrosti prenosa predvideli največ en dodaten bajt. Novejši krmilniki RS-232 s 16-bajtnim internim izravnalnikom pa v takšnem primeru pošljejo do 16 bajtov (tj. celotno vsebino internega izravnalnika). Zato ima Sinclairjeva uradna koda v Mavrici težave že s prenosi, večjimi od 2.400 baudov. YX uporablja 16-bajtni izravnalnik, in nima težav pri komunikaciji s sodobnimi krmilniki RS-232.

9.3.3 Bitna eksplozija in večopravilnost

Med pošiljanjem/sprejemanjem dveh zaporednih bitov se zanašamo na hitrost procesorja in število t-stanj posamičnega strojnega ukaza. Zato programa za pošiljanje/sprejem ne smemo prekinjati.

Upoštevanje tega pravila bi bilo neboleče pri pošiljanju manjših količin podatkov. Pri večjih količinah pa bi začelo ovirati večopravilnost. Procesi bi postali neodzivni, dokler vseh podatkov ne bi prenesli. Rešitev težave je v razbitju celotne količine podatkov na več manjših blokov, ki jih pošljemo brez prekinitve. V vmesnem času pa podatkov ne pošiljamo in sprejem zaustavimo z uporabo signala CTS ter prepustimo procesorski čas tudi drugim opravilom.

9.3.4 Efektivna hitrost prenosa v večopravilnem okolju in mrtva izguba.

Mavričina prekinitev se sproži petdesetkrat v sekundi oziroma »približno« vsakih 69.888 t-stanj. Če branje enega bajta z zaporednih vrat traja »približno« 300 t-stanj, to pomeni, da N-bajtov lahko preberemo v N * 300 t-stanjih.

Pri hitrosti 115.200 ob branju enega bita neproduktivno porabimo »približno« 8 t-stanj (dva strojna ukaza ***NOP***) oziroma 64 t-stanj na bajt. Če vsako prekinitev preberemo 64 bajtov, smo neproduktivno porabili 4.096 od porabljenih 19.200 t-stanj, za vsa druga opravila v sistemu (branje tipkovnice, utripanje zaslonskega kazalca, preklapljanje med procesi) pa jih ostane 50.688. Sistem med zaporednimi prenosi ostane relativno odziven. Po tem scenariju je efektivna hitrost prenosa 64 bajtov v 1/50 sekunde oziroma 3.200 bajtov na sekundo.

Formuli za izračun približne efektivne hitrosti prenosa in števila stanj, ki so na voljo preostalim procesom v odvisnosti od števila bajtov, ki naj se preberejo med prekinitvijo, sta

$$procesorski\ čas\ za\ druga\ opravila = 69.888 - (število\ bajtov * 300)$$

$$efektivna\ hitrost\ v \frac{bajtih}{sekundo} = število\ bajtov * 50$$

Število bajtov, ki naj se preberejo med prekinitvijo, in s tem razmerje med odzivnostjo sistema in efektivno hitrostjo prenosa podatkov, je nastavljivo.

9.3.5 Izvedba hitrega prenosa na Mavrici

Pošiljanje s hitrostjo 115.200 baudov je relativno preprosto. Potrebno je le v pravilnih časovnih razmikih spreminjati signal na vratih ***0xef***. Pri sprejemu pa naletimo na dve časovni stiski – pri zaznavanju začetnega bita in pri shranjevanju sprejetega bajta.

9.3.6 Zaznava začetnega bita

Zaznavo je treba opraviti tako, da je predvidljiva v najboljšem in najslabšem primeru. Če začetnega bita ni, je po nekaj ponovitvah potrebno prekiniti sprejem.

Pogosto uporabljani ukazi in zanke procesorja Z80 tega ne omogočajo. K sreči pa je sedmi bit, ki ga Mavrica uporablja kot podatkovni bit za RS-232, hkrati tudi predznak bajta. Kadar je nastavljen, je nastavljena tudi zastavica ***M***. Če uporabimo takšen ukaz za branje vhodnih vrat, ki spreminja zastavice, lahko začetni bit zaznamo z le dvema ukazoma. Da pa stvari ne bi bile preveč enostavne, drugi ukaz ne sme biti klasičen pogojni skok, saj so ti časovno preveč potratni. Pri izvedbi želenega skoka in testa si pomagamo z ukazom za vrnitev iz podprograma ret, pred tem pa še ustrezno nastavimo sklad.

```
                ;; initial values
                ld      a,#CTS_ON               ; xxx1xxxx - CTS
                ld      b,#RS232_IBUFF_SIZE     ; b=buffer size
                ld      c,#RS232_DTA            ; c=data port
                ld      de,#0x8000              ; masks

                ;; intialize SP
                ld      (#_rs232_sp),sp         ; store SP
                ld      sp,#retm_table          ; 20 x same return address

                ;; announce clear to send (CTS)
                out     (RS232_CTL),a           ; 11 t-states

start_bit:
                ;; get start bit, try 10 times before giving up
                in      a,(c)                   ; read rs232 bit / 12 t-states
                ret     m                       ; if start bit / 5/11 t-states
                in      a,(c)
                ret     m
                in      a,(c)
                ret     m
                in      a,(c)
                ret     m
                in      a,(c)
                ret     m
                in      a,(c)
                ret     m
                in      a,(c)
                ret     m
                in      a,(c)
                ...
retm_table:
```

Skok na ustrezno mesto ob vrnitvi pa zagotovi ustrezno stanje sklada.

```
retm_table:
                .word data_bits, data_bits, data_bits, data_bits
                .word data_bits, data_bits, data_bits, data_bits
```

```
                .word data_bits, data_bits, data_bits, data_bits
                .word data_bits, data_bits, data_bits, data_bits
```

Ker bi bilo štetje ponovitev poskusov zaznave začetnega bita časovno predrago – poskuse enostavno ponovimo. Tako v najboljšem primeru v 11 urnih periodah in v najslabšem primeru v 28 urnih periodah zaznamo startni bit. Razlika med najslabšim in najboljšim primerom znaša 17 urnih period, kar je dovolj zanesljivo.

9.3.7 Shranjevanje sprejetega bajta

Zaradi potrebe po 16-bajtnem izravnalniku iz poglavja RTS/CTS se ni mogoče zanesti na CTS, da bo zaustavil pošiljatelja, ampak je treba biti pripravljen sprejeti vsaj 16 bajtov zaporedoma, četudi se že po prvem sprejetem bajtu s CTS drugi strani sporoči, naj počaka. To pa bi pomenilo, da bi nam znova zmanjkalo časa za zanesljivo branje. K sreči protokol RTS/CTS pozna nastavitev, po kateri pošlje dva končna bita. To pa po zaznanju zadnjega bita omogoča še dodatnih 60 urnih period, kar je dovolj za shranjenje sprejetega bajta in skok na kodo za sprejem novega.

Koda prikazuje sprejem do 20 bajtov prek zaporednih vrat RS-232.

```
data_bits:
                ;; add 21 t-states
                cp      #0                      ; 7 t-states
                dec     a                       ; 4 t-states
                nop                             ; 4 t-states
                nop                             ; 4 t-states

                ;; start reading data bits
                in      a,(#0xf7)       ; bit 0  | 11 t-states
                rla                             ; lsb to carry / 7 t-states
                rr      e                       ; and to msb (e) /8 t-states
                nop                             ; 4 t-states

                in      a,(#0xf7)
                rla
                rr      e
                nop

                in      a,(#0xf7)
                rla
                rr      e
                nop

                in      a,(#0xf7)
                rla
                rr      e
                cp      #0                      ; waste extra cycles from now on

                in      a,(#0xf7)
                rla
                rr      e
```

```
                cp      #0


                in      a,(#0xf7)
                rla
                rr      e
                cp      #0

                in      a,(#0xf7)
                rla
                rr      e
                cp      #0

                in      a,(#0xf7)
                rla
                rr      e

                ;; t-states of 2 stop bits ... to prepare for next start bit
                ld      a,e                     ; 4 t-states
                cpl                             ; 4 t-states
                ld      (hl),a                  ; write to buffer | 7 t-states
                inc     hl                      ; increase buff. counter | 6 t-
states

                ;; tell sender to stop sending (every time you read a byte...)
                ld      a,#CTS_OFF              ; xxx0xxxx (set cts) | 7 t-states
                out     (RS232_CTL),a           ; raise CTS | 11 t-states

                djnz    jp_start_bit            ; 13/8 t-states

                ld      sp,(#_rs232_sp)         ; restore stack

                pop     de                      ; original buffer to de
                or      a                       ; clear carry flag
                sbc     hl,de                   ; count of bytes to hl

                call    _ei
                ret

jp_start_bit:   jp      start_bit
```

Poglavje 10 Prenaslovljivost

10.1 Kaj je prenaslovljiv program?

Prenaslovljiv program lahko naložimo na katerikoli pomnilniški naslov in se bo pravilno izvedel. To pomeni, da morajo biti bodisi vsi skoki in sklici na podatke v programu relativni ali pa je treba ob nalaganju programa izvesti prenaslavljanje, tj. ponovno izračunati vse skoke in sklice na podatke.

10.2 Prevajanje za prenaslavljanje

Povezovalnik SDCC ne zna generirati prenaslovljivih programov. Vnaprej mu je treba povedati, na katerem pomnilniškem naslovu se bodo izvajali. Na procesorju, ki je brez enote za upravljanje s pomnilnikom, in v okolju, kjer hkrati teče več opravil, to predstavlja resno oviro. Ni namreč mogoče zagotoviti, da bo določena pomnilniška lokacija vselej prosta za program.

Obe očitni rešitvi tega problema sta bili zavrženi – 1) napisati nov povezovalnik bi zahtevalo preveč časa, 2) uporabiti šibkejši prevajalnik z88dk, ki zna proizvesti kodo, ki se zna sama premestiti, pa bi pomenilo prevelik korak nazaj.

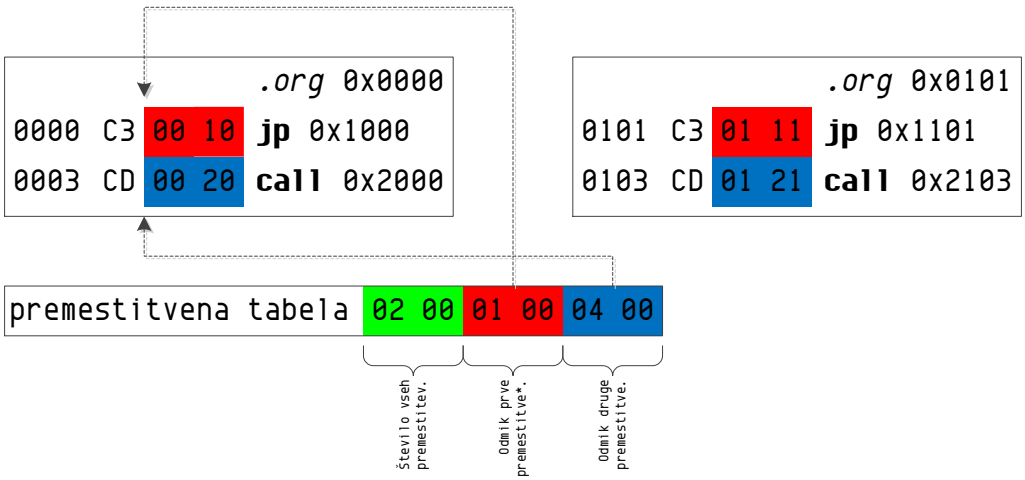
Izbrana rešitev je mnogo preprostejša, vendar nekoliko tvegana. Zahteva defenzivno programiranje. Uporabi pa se lahko vedno, kadar je potrebno generirati prenaslovljiv program s prevajalnikom, ki tega ne zna.

Program se s povsem enakimi nastavitvami prevede na dva različna naslova. Potem se obe programski datoteki primerjata. Zaznane razlike med njima so naslovi, ki jih je razrešil povezovalnik in jih je treba pri prenaslavljanju programa na novo izračunati.

Te razlike se zapišejo v prenaslovitveno tabelo in dodajo v glavo programske datoteke. Ob nalaganju programa se jedro najprej sprehodi skozi prenaslovitveno tabelo in ponovno izračuna vse skoke in sklice na podatke ter ustrezno popravi program, šele nato pa skoči na njegov začetek.

10.3 Prenaslovitvena tabela

Prva beseda[7] v prenaslovitveni tabeli je število vnosov v njej. Vsaka naslednja beseda pa je kazalec (odmik od začetka programa) na mesto v programu, kjer je shranjen podatek, ki ga je potrebno ob prenaslavljanju ponovno izračunati.



Slika 17. Kako nastane premestitvena tabela.

Zgornja slika prikazuje nastanek prenaslovitvene tabele. Isto datoteko prevedemo prvič na naslov ***0x0000*** in drugič na naslov ***0x0101***. Obe programski datoteki primerjamo. Na sliki so z rdečo in modro barvo označena mesta, kjer se vsebina obeh razlikuje.

Ker je v Z80 naslov dolg 16 bitov, se med obema datotekama vedno razlikujeta po dva zaporedna bajta.

Prva beseda v prenaslovitveni tabeli vsebuje vrednost 2[8], kar je število besed, ki sledijo. Naslednji dve besedi v prenaslovitveni tabeli pa sta kazalca na ustrezno mesto v programu.

Pri eksperimentiranju se je pokazalo, da je pomembno, da je drugi naslov, na katerega prevedemo program, lih, tj. ***0x0101***. V primeru sodega naslova, npr. ***0x8000***, bi se manj pomembna bajta v kodi obeh programov lahko ujemala, in ker Z80 uporablja konvencijo *little endian*, ne bi bilo mogoče zaznati razlike v datotekah (pri obeh bajtih, ki sestavljata naslov).

Za generiranja prenaslovitvene tabele je bil napisan poseben program relocator[9].

[7] Beseda je 16-biten podatkovni tip.
[8] Po konvenciji little endian zaporedna bajta 0x02 0x00 tvorita besedo 0x0002.
[9] Izvorna koda programa je na repozitoriju GIT.

10.4 Prenaslavljanje

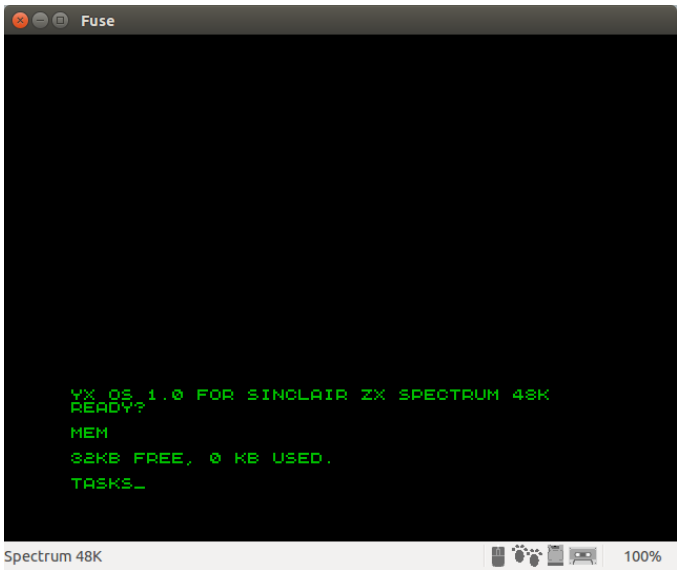
Prenaslovitvena tabela z metapodatki je del glave programske datoteke. Sledi program, ki je preveden na naslov ***0x0000***. Jedro ob nalaganju najprej prebere osnovne metapodatke o opravilu (zahtevana velikost sklada, velikost programa itn.) ter rezervira pomnilnik zanj. Potem pa na naslov naloži program in se s pomočjo prenaslovitvene tabele »sprehodi« čezenj in opravila prenaslovi.

10.5 Pravila defenzivnega programiranja

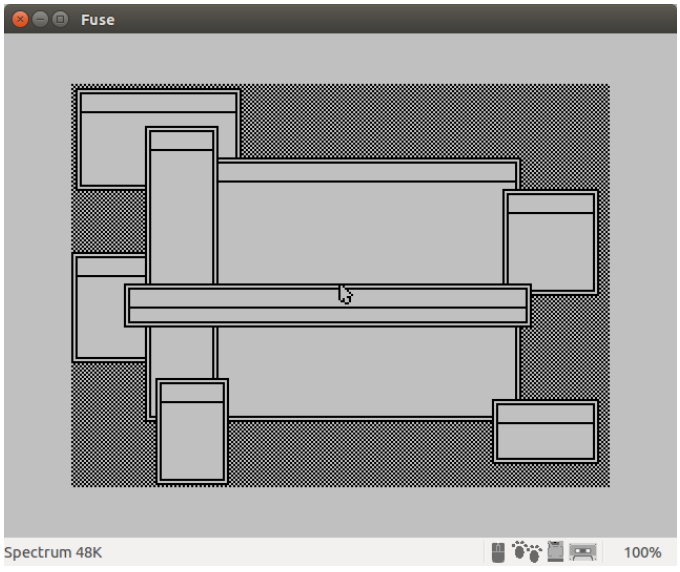
Da bi prenaslavljanje pravilno delovalo, je treba upoštevati pravila defenzivnega programiranja. Programi v zbirniku ne smejo uporabljati absolutnega naslavljanja, razen na znane naslove. Ne smejo uporabljati poravnave kode (npr. na 256-bajtno mejo) ali trikovizračunavanje nerelativnih skokov z uporabo ***JP HL***.

Poglavje 11 Primera uporabe mikrojedra YX

Kot ilustracijo delovanja mikrojedra in ne kot del naloge smo razvili še dva preprosta programčka: lupino in zametek sodobnega grafičnega uporabniškega vmesnika.



Slika 18. Preprosta lupina



Slika 19. Zametek preprostega grafičnega uporabniškega vmesnika

Poglavje 12 Sklepne ugotovitve

Med pisanjem diplomske naloge smo prišel do dveh sklepnih ugotovitev.

Mavrica je bila dovolj močna strojna platforma, da bi lahko na njej poganjali več opravil hkrati. Pri delu bi lahko uporabljali grafični uporabniški vmesnik, primerljiv s tistim s prvega Macintosha. In bili prek serijske povezave priključeni v lokalno omrežje. Da tega nismo mogli, niso bile krive (le) omejitve strojne opreme, ki jih je narekovala ekonomika, ampak predvsem dejstvo, da je bilo v času načrtovanja Mavrice sodobno znanje o operacijskih in okenskih sistemih še v povojih ali pa še ni obstajalo.

Prva sklepna ugotovitev naloge je, da je mogoče zgolj z novimi znanjem in brez sprememb strojne opreme (oz. na izumrli strojni opremi) zgraditi in poganjati sodobne programe.

Druga sklepna ugotovitev je, da je razvoj jeder na sistemih z omejenimi viri zelo dober pedagoški pripomoček za praktično učenje o operacijskih in vgradnih sistemih.

Zaradi preprostosti strojne opreme je lažje razumeti rešitve zanjo. Jedra je mogoče poenostaviti v tolikšni meri, da jih lahko razvijemo v enem semestru. Prav takšnega projekta so se, denimo, lotili na univerzi Cambridge, kjer kot učni pripomoček in kot javna storitev (ang. community service) nastaja projekt, podoben jedru YX – za priljubljeni Raspberry PI. Več o tem lahko preberete na njihovih spletnih straneh

https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/

Poglavje 13 Literatura

13.1 Seznam virov

[1] Helen Custer, Inside Windows NT, Microsoft Press, 1993.

[2] Marko Batista, Ciciban, številka 3, 1984.

[3] ZiLog Worldwide, Z80 CPU User's Manual, dostopno na: http://home.mit.bme.hu/~benes/oktatas/dig-jegyz_052/Z80.pdf.

[4] Jorrit N. Herder, Towards a True Microkernel Operating System, Vrije Universiteit Amsterdam, 2005.

[5] Marwan Burelle, A Malloc Tutorial, Laboratoire Système et Sécurité de l'EPITA (LSE), 2009.

[6] Arpaci-Dusseau, Remzi H.; Arpaci-Dusseau, Andrea C., Operating Systems: Three Easy Pieces, Arpaci-Dusseau Books, 2014.

[7] The SDCC compiler user's guide, dostopno na: http://sdcc.sourceforge.net/doc/sdccman.pdf.

[8] Jack Ganssle, Bit Banging for the RS232, The Ganssle Group, dostopno na: http://www.ganssle.com/articles/auart.htm.

[9] SDCC vs z88dk: Comparing size and speed of the binaries generated for Amstrad CPC, dostopno na: http://www.cpcmania.com/Docs/Programming/SDCC_vs_z88dk_Comparing_size_and_speed.htm.

[10] Wischner, Blog o programiranju za ZX Spectrum, dostopno na: http://wischner.blogspot.co.uk/search/label/zx%20spectrum.